VŠB – Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Knihovna pro modelování a simulace

Library for Modelling and Simulation

2013 Bc. Radek Scholz

VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Zadání diplomové práce

Student: **Bc. Radek Scholz**

Studijní program: N2647 Informační a komunikační technologie

Studijní obor: 2612T025 Informatika a výpočetní technika

Téma: Knihovna pro modelování a simulace
Library for Modelling and Simulation

Zásady pro vypracování:

Cílem práce je vytvořit knihovnu pro vytváření modelů a simulací a to jak diskrétních tak i spojitých. Knihovna bude sloužit jako platforma pro tvorbu obecných simulací a jako podpora v předmětu Modelování a simulace.

Práce bude obsahovat:
1. Rešerši existujících knihoven.
2. Popis potřebných komponent pro diskrétní a spojité simulace.
3. Návrh knihovny a její implementace.
4. Otestování knihovny na názorných příkladech.

Knihovna bude realizována v jazyce C# pomocí .Net technologie.

Seznam doporučené odborné literatury:

[1] R. Pelánek: Modelování a simulace komplexních systémů, Nakladatelství Masarykovy univerzity, 2011, ISBN: 978-80-210-5318-2
[2] Jay Glynn a kol.: C# Programujeme profesionálně, COMPUTER PRESS, ISBN: 9788025100851

Formální náležitosti a rozsah diplomové práce stanoví pokyny pro vypracování zveřejněné na webových stránkách fakulty.

Vedoucí diplomové práce: **Ing. Jan Platoš, Ph.D.**

Datum zadání: 16.11.2012
Datum odevzdání: 07.05.2013

doc. Dr. Ing. Eduard Sojka
vedoucí katedry

prof. RNDr. Václav Snášel, CSc.
děkan fakulty

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

V Ostravě 30. dubna 2013 .....

Chtěl bych vyjádřit poděkování vedoucímu mé diplomové práce Ing. Janu Platošovi Ph.D. za jeho čas , který se mnou strávil konzultacemi na této diplomové práci. Rovněž bych chtěl poděkovat za jeho odborné rady, cenné připomínky a ochotu, bez kterých by tato diplomová práce nevznikla.

Abstrakt

Cílem této práce je vytvořit knihovnu pro modelování a simulace. Tato knihovna bude umět simulovat jak diskrétní , tak spojitou simulaci. Knihovna bude naprogramována v programovacím jazyce C#.

Klíčová slova: C#, diskrétní , spojitá, simulace

Abstract

The aim of this work is to create a library for modeling and simulation. This library will be able to simulate both discret and continuous simulation. Library is programmed in programming language C# .

Keywords: C #, discrete, continuous, simulation

Seznam použitých zkratek a symbolů

SHO	–	Systém hromadné obsluhy

Obsah

Seznam obrázků

Seznam tabulek

Seznam výpisů zdrojového kódu

1 Úvod

V dnešní době se s termínem simulace setkáváme v různých souvislostech. Termín simulace můžeme chápat v různých pojetích, ať už to jsou mimo jiné počítačové simulátory, simulátory pro výuku a výcvik a spoustu dalších. Avšak většina lidí si pod pojmem simulace představí napodobování skutečných věcí v reálném světě.

Tato diplomová práce se zabývá simulací počítačovou. To tedy znamená, že se zde jedná o napodobení určitého reálného systému pomocí počítačového modelu.

Cílem simulace je získání nových informací a znalostí s experimentováním s jeho modelem. U modelu můžeme jednoduše změnit jeho parametry, respektive jeho vlastnosti a následným spuštěním simulace zjistit, jak tyto vlastnosti ovlivní chování celého systému. Simulace nám poskytne takové informace, které pak můžeme zužitkovat při návrhu nového reálného systému. Tyto informace můžou výrazně ovlivnit budoucí podobu tohoto systému.

Nesmíme zapomenout na skutečnost, že simulace je pouze jakýmsi odhadem charakteristik z jeho reálného systému. K podrobnějšímu vysvětlení termínu simulace se dostaneme v druhé kapitole.

Tato diplomová práce je zaměřena na dva základní typy simulace. A to simulaci diskrétní, na kterou byl kladen největší důraz a je tedy nejdůležitější součástí této diplomové práce, ale také simulací spojitou. Diskrétní simulace je podrobně vysvětlena v třetí kapitole, kde se čtenář seznámí se základními principy, pojmy a vlastnostmi této simulace. Čtvrtá kapitola je věnována samostatné knihovně **DiscreteSimulationLibrary**. Jak už nám název napovídá, bude se jednat o knihovnu zprostředkující diskrétní simulaci. Dále si v této kapitole vysvětlíme, jak jednotlivé třídy a metody, kterýma tato knihovna disponuje, tak návod jak správně tuto knihovnu využívat. V páté kapitole se čtenář blíže seznámí se simulací spojitou. Kapitola šestá popisuje knihovnu **ContinuousSimulationLibrary**, která slouží pro modelování a simulaci spojitých systémů. Podobně jako u knihovny pro diskrétní simulace si popíšeme veškeré vlastnosti, třídy, metody a použití této knihovny.

2 Modelování a simulace

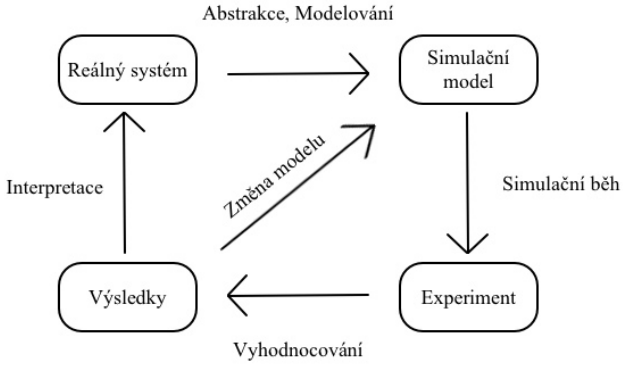
2.1 Simulace

V úvodu diplomové práce jsme si trochu popsali pojem simulace, ale až nyní si termín simulaci přiblížíme o něco více. Definujeme si termín simulace, především tedy simulaci počítačovou, řekneme si jaké má vlastnosti, přednosti a nevýhody.

Jak už bylo jednou napsáno termín simulace znamená napodobovat či imitovat reálné objekty, stavy nebo také různé procesy. V širším pojetí se s tímto pojmem můžeme setkat v různých souvislostech. Patří zde modelování lidských systémů, jako příklad můžeme uvést hodně známé letecké simulátory nebo trenažéry v autoškolách. Simulace dále může zahrnovat modelování přírodních systémů, technologické simulace pro optimalizaci výkonu, simulace pro bezpečnostní inženýrství, simulace pro testování, simulace pro školení a vzdělávání a spoustu dalších. V užším pojetí, kde výpočet neznámých parametrů a napodobování reálných systému probíhá na počítači, označujeme simulaci jako počítačovou, kterou budeme v tomto textu dále popisovat.

Termín simulace je v různých literaturách definován různě. Například Shannon definuje simulaci takto:

Definice 1 *Simulace je proces tvorby reálného systému a provádění experimentů s tímto modelem za účelem dosažení lepšího pochopení studovaného systému či za účelem posouzení různých variant činnosti systému. [1]*



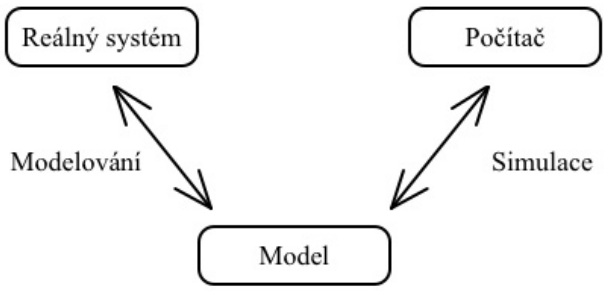
Obrázek 1: Diagram dle Shannona

Dle Naylora může znít definice takto:

Definice 2 *Numerická metoda, která spočívá v experimentování s matematickými modely (dynamickými) reálných systému na číslicových počítačích. [1]*

Nebo dle Zeiglera:

Definice 3 *Tři elementy (reálný systém, model, počítač) a dva vztahy, modelový a simulační, jak je naznačeno na obr.2[1]*



Obrázek 2: Diagram dle Zeiglera

2.1.1 Základní časové pojmy

U simulace musíme vymezit tři základní časové pojmy:[6]

a) **Reálný čas** – čas, ve kterém probíhá skutečný děj v reálném systému.

b) **Simulární čas** – čas, jež tvoří časovou osu modelu. Může být reálný, zrychlený, zpomalený. Přičemž musí splňovat následující podmínky:[7]

- hodnota simulárního času nesmí v průběhu výpočtu klesat
- děje v simulačním modelu závisejí na simulárním čase stejným způsobem, jako jejich vzory ve výchozím systému na toku přirozeného času.

Simulární čas by neměl být uživateli přístupný. Uživatel však může hodnotu simulárního času zjistit (např. nějakou metodou), ale nemůže tento čas měnit.

c) **Strojový čas** – jedná se o čas spotřebovaný procesorem pro výpočet programu. Tento čas se odvíjí od složitosti modelovaného systému a na vlastnostech programu. Tento čas není závislý na hodnotách simulárního času.

2.1.2 Výhody a nevýhody simulace

Mezi výhody simulace patří:

- simulace nám umožňuje prověřit různé varianty nastavení simulovaného systému, tím je myšleno u počítačové simulace změna parametrů, bez nutnosti vynaložení zdrojů na její realizaci
- simulace umožňuje změnu rychlosti času, tuto vlastnost můžeme využít k prozkoumání jevu, který by trval například několik hodin, ale díky simulace můžeme tento jev prozkoumat během několika sekund, nebo v opačném případě, jev který trvá řádově několik milisekund, může zpomalit a detailně prozkoumat
- simulací lze řešit složité systémy, které nelze řešit analytickými metodami
- simulace nabízí komplexnější pohled na studovaný problém
- pomocí pozorování simulačního modelu, lze lépe pochopit reálný systém

Mezi nevýhody simulace patří:

- problém při vytváření modelu, špatně navržený model, může mít za následek chybný výsledek simulace
- u některých simulací lze obtížné interpretovat výsledky, protože výstupy jsou v podstatě náhodné veličiny vzhledem k náhodným vstupům, proto může být někdy obtížné rozlišit, zda některá pozorování jsou důsledkem vnitřních vztahů v systému nebo důsledkem náhody v simulačních experimentech
- simulační modelování a analýzy mohou být časově náročné a drahé

2.1.3 Dělení simulace

Simulaci můžeme rozdělit hned dle několika charakteristických kategorií do podskupiny. Jestliže se podíváme na simulaci z hlediska jeho modelu, kde rozhodujícím faktorem bude čas, dělíme simulaci na **spojitou** a **diskrétní**.

Popíšeme-li chování nějakého reálného systému počítačovým modelem a budou nás zajímají všechny změny v každém okamžiku, označujeme tuto simulaci za spojitou. Příkladem využití spojité simulace je simulace pro zkoumání různých fyzikálních procesů, u kterých dochází ke změnám kontinuálně.

Ne vždy nás zajímají všechny změny v systému. V těchto případech nás zajímají pouze změny při určitých událostí. To znamená, že v reálném systému stále dochází ke změnám mezi těmito událostmi, ale z hlediska simulace jsou pro nás tyto změny nepodstatné a nezajímavé. Takovouhle simulaci označujeme za simulaci diskrétní. U diskrétní simulace tedy zaznamenáváme pouze zajímavé a relevantní události a časové okamžiky, ve kterých tyto události nastávají. Typickým příkladem použití diskrétní simulace je systém hromadné obsluhy.

Simulace, které mají vlastnosti typické pro spojitou simulaci, tak vlastnosti typické pro diskrétní simulaci, nazýváme kombinovaně diskrétně-spojité nebo častěji jednoduše kombinované simulace.

Simulace se také dělí podle druhu simulátoru. Na analogovém nebo číslicovém počítači, Real-Time simulace, paralelní a distribuovaná simulace.

Další možnosti dělení:[2]

- vnořená simulace
- interaktivní simulace
- virtuální realita
- a další

2.2 Systém

Je to určitá množina prvků, které mají mezi sebou určité vazby. Definování systému obsahuje zjednodušený a zobecněný model, ale i tak přesný popis důležitých vlastností prvků a vazby mezi nimi.
Vazby dělíme na:

1. **Vnitřní** - jedná se o vazby mezi jednotlivými prvky systému
2. **Vnější** - vazby mezi prvky systému a jeho okolím

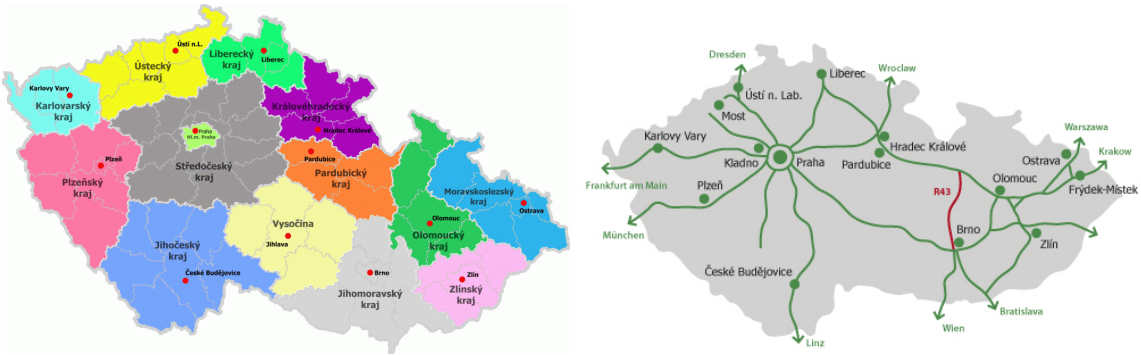
2.3 Model

Model je zjednodušenou a zobecněnou imitací reality.

> *Všechny modely jsou špatně. Některé modely jsou užitečné* (G. Box, W. E. Deming)

První věta tohoto citátu nám jasné říká, že model nemůže být nikdy úplně dobře. Vždy se liší od reality. Druhá věta nevylučuje fakt, že i když je tento model zjednodušený a zobecněný neznamená to, že nám nemůže být užitečný. Ale musíme vzít v úvahu, že ne všechny modely jsou užitečné.

Uveďme si konkrétní příklad, který všichni dobře známe: mapa. Mapa je model prostoru. Mapa je špatně, protože je to abstrakce reality: neobsahuje všechny detaily a dochází u ní ke zkreslení. Jak každý z vlastní zkušenosti jistě potvrdí, sebelepší mapa má řadu chyb. I přesto je mapa velmi užitečná, a to z mnoha důvodů: pochopení reality (mapa jako výuková pomůcka), plánování akcí (kterou cestou se mám vydat) nebo usnadnění komunikace (potkáme se na tomto místě).[4]



(a) Kraje ČR (b) Dálnice ČR

Obrázek 3: Ukázka map

2.4 Modelování

Modelování je jakýmsi postupem jak vytvářet zjednodušený obraz reálného systému. Tomuto obrazu říkáme model, jak už jsme si vysvětlili v předchozím odstavci. Není prakticky možné vytvořit přesný popis modelu, protože jednotlivé reálné systémy jsou velice komplikované. Proto při modelování musíme vybírat pouze podstatné části a nepodstatné části opomíjet. Ten kdo vytváří model musí sám rozhodnou zda se jedná o podstatnou či nepodstatnou věc. Z tohoto důvodu nelze proces modelování algoritmizovat.

Smyslem modelování je náhrada zkoumaného systému jeho modelem a pomocí experimentů s modelem získat informaci o původním zkoumaném systému. Výsledkem modelování je tedy vytvořený model.[3]

Jeden z hlavní principů při modelování zní: *Nemodelovat systém, modelovat problém.* Aby byl model užitečný, musí mít jasný účel. Pokračujeme s příkladem s mapou. Máme různé mapy, každou pro specifický účel (např. automapy, cyklomapy, turistické mapy, vodácké mapy), univerzální mapa by byly k ničemu. Musíme si jasně zvolit účel modelu a potom se snažit udržovat model co nejjednodušší.[4]

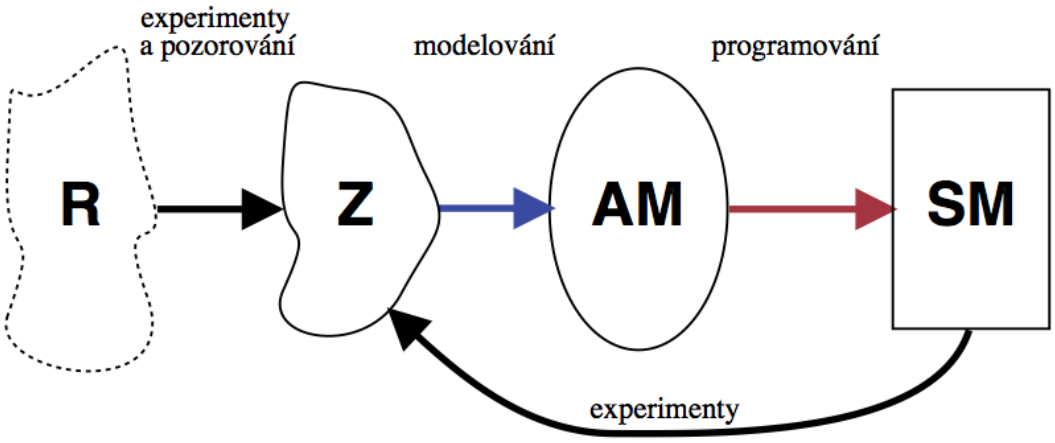
2.4.1 Zjednodušený proces modelování

Proces modelování můžeme zjednodušeně rozdělit do tří základních etap.

1. Vytvoření abstraktního modelu - formování účelového a zjednodušeného popisu zkoumaného systému.
2. Vytvoření simulačního modelu - zapsání abstraktního modelu formou programu.
3. Simulace - experimentování s reprezentací simulačního modelu na počítači.

K podrobnějšímu popisu procesu modelování se dostaneme v dalších kapitolách.

REALITA → ZNALOSTI → ABSTRAKTNÍ MODEL → SIMULAČNÍ MODEL



Obrázek 4: Zjednodušený princip modelování a simulace

Pramen: PERINGER, P.: Modelování a simulace

2.5 Fáze modelování

Proces modelování lze rozdělit do šesti základních fází. Tyto fáze jsou idealizované, to znamenám, že při tvorbě modelu procházíme jednotlivé fáze iterativně, ale ve skutečnosti se podle potřeby vracíme k předchozím bodům.

1. Formulace problému
2. Základní návrh modelu
3. Implementace modelu
4. Verifikace a validace
5. Simulace a analýza
6. Sumarizace výsledků

2.5.1 Formulace problému

Při formulaci problému se budeme řídit pravidlem: *nemodelovat systém, modelovat konkrétní problém*. V kapitole 2.4 jsme se sice s tímto pravidlem už seznámili, ale teď si tohle pravidlo vysvětlíme detailněji. Jak už samotný název této podkapitoly napovídá, jedná se o formulaci problému, který se snažíme modelovat. To znamená, že se budeme snažit co nejpřesněji a nejkonkrétněji popsat modelovaný systém modelem. Musíme jasně formulovat, jaký konkrétní problém řešíme a co se pomocí tohoto modelu chceme dovědět. U formulace problému si musíme jasně určit časový horizont, jestli se bude jedná o minuty, hodiny, roky, století atd. Dále musíme brát v úvahu jaké máme prostředky.

Jako příklad si můžeme zvolit téma „počasí“. Když se podíváme na problém počasí, musíme si jasně určit co nás konkrétně zajímá (druhá věta ze základního principu modelování: *Nemodelovat systém, modelovat konkrétní problém*), jestli nás zajímá počasí na dnešní den, zvolíme časový horizont řádově hodiny. Jestliže budeme chtít modelovat počasí v jednotlivých ročních obdobích, zvolíme časový horizont v řádu několika měsíců.

2.5.2 Základní návrh modelu

Model se snažíme tvořit co nejjednodušeji, tj. zaznamenat ho v co největší abstrakci, a až poté přidávat nejrůznější detaily. Při základním návrhu modelu si musíme určit jednotlivé okraje modelu. To znamená, že si určíme jaké prvky budeme modelovat.

Zaměřujeme se především na okraje „ do šířky “, tj. co vše bude v modelu zohledněno, rámcově určujeme také okraje „ do hloubky “, tj. jak detailně budou jednotlivé prvky zohledněny.[4] U tvorby modelu vybíráme jeho hlavní prvky, části a v neposlední řadě vztahy mezi nimi. U vztahu určuje co s čím souvisí, tzv. kvalitativní vztahy, nikoli jak přesně to souvisí, tzv. kvantitativní vztahy.

Pro názornou ukázku můžeme pokračovat v tématu „ počasí “. Náš časový horizont bude zvolen v řádu měsíců. V tomto konkrétním případě by mezi hlavní části modelu patřilo na jakém kontinentu zkoumáme počasí a jaké je roční období. Tyto vymezující oblasti nám určují okraje do šířky. Vymezení do hloubky znamená, jak detailně popíšeme jednotlivé prvky. Nesmíme zapomenout popsat jednotlivé vztahy. Jako příklad můžeme uvést vztah: jestliže je země a konkrétní kontinent, u kterého zkoumáme počasí blíže ke slunci, znamená to, že zde bude tepleji. Chceme mít v modelu i opačnou vazbu, tj. ovlivní vzdálenost země ke slunci teplota vzduchu.

2.5.3 Implementace modelu

Implementace modelu může mít různé přístupy a nástroje pro modelování, proto si musíme vybrat, jak budeme modelovat a jaký nástroj použijeme. Dokončíme a doplníme všechny části modelu. Například počítačový model, aby byl spustitelný, musí být kompletní a musí jasně formulovat všechny předpoklady.

Když už máme model takhle připravený, bude posledním krokem implementace modelu zadání veškerých hodnot u parametrů a doplnění kvantitativních informací. Tyto informace určujeme na základě pozorování, statistických měření nebo prostého odhadu.

2.5.4 Validace a verifikace

Validace a verifikace nám spolu ověřují, zda námi vytvořený model je dobře navržený a je schopen nám odpověď na naše otázky . Jedná se o jeden z nejobtížnější kroku při tvorbě modelu.

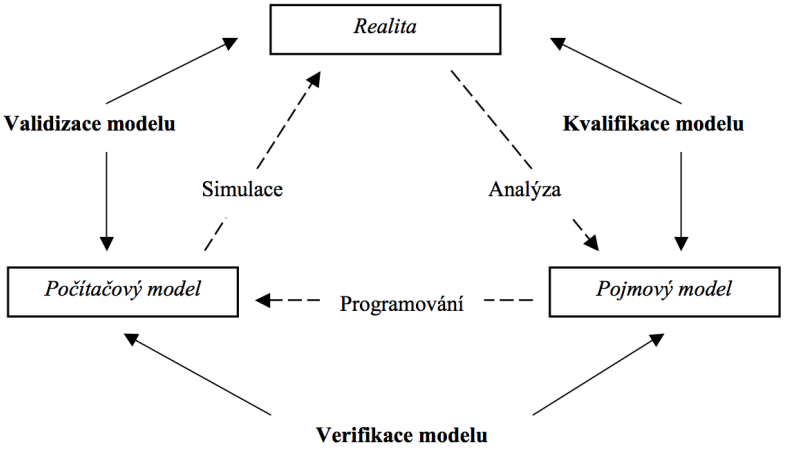
Validace

Ověření, zda navrhovaný abstraktní model odpovídá chování reálného systému. Neexistuje všeobecný postup, jak by se měla validace modelu řídit. Za nejběžnější způsob se považuje připravit několik typických simulačních experimentů, pomocí kterých by se model prověřil za určitých podmínek.

Pro názornou ukázku budeme pokračovat v tématu s počasím. Můžeme ověřit, že když vypneme vztah, kdy země je dále od slunce, pak dojde k ochlazení teploty.

Verifikace

Verifikace je ověření, že model opravdu dělá to, co si myslíme, že by měl dělat, tj.ověření vztahu mezi abstraktním návrhem modelu a jeho konkrétní implementací.[4]



Obrázek 5: Ověření věrohodnosti modelu

Pramen: KŮS, Z.: Simulace

2.5.5 Simulace a analýza

Námi vytvořený model musíme analyzovat, musíme si však být jistí, zda je pro naše účely dobře navržen. Pro jednotlivé konkrétní příklady máme různé cíle analýzy.

Typicky však spadají pod následující otázky. Jakou roli hrají jednotlivé prvky modelu? Které prvky modelu mají největší vliv na jeho chování? Jaké je chování modelu za změněných (měnících se) podmínek?[4]

2.5.6 Sumerizace výsledků

Výsledky by nám měly dát odpovědi na naše otázky. V případě potřeby se vracíme k jednotlivým bodům.

Případné otázky:[4]

- Podařilo se najít odpovědi na původní otázky?
- Splňuje model účel?
- Plynou z modelování a simulace nějaké závěry a ponaučení? Jaké?
- Jak můžeme model dále využívat?
- Je potřeba model rozšířit? Proč? Jaké rozšířeni by bylo vhodné?

2.6 Cíl modelování a simulace

Cíl modelování a simulace není jednoznačně určen, protože modelování a simulaci můžeme použít pro mnohé účely.

Návrh a řízení systémů

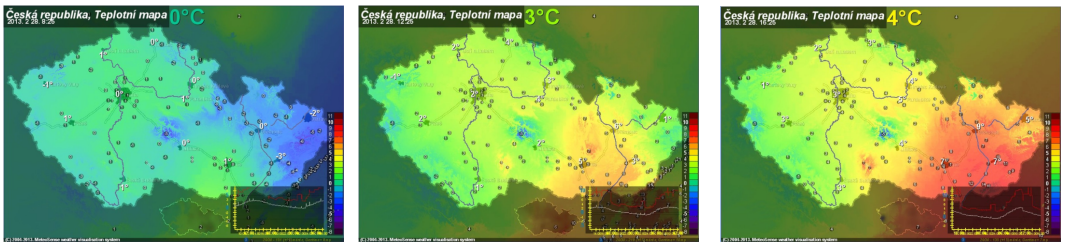
Jestliže sami navrhujeme nějaký systém, můžeme využít simulaci k tomu, abychom mohli vyzkoušet jednotlivé varianty a provést různé změny nastavení. To vše za předpokladu, že navrhovaný systém podrobně známe a máme nad ním výraznou kontrolu. Při tomto návrhu se snažíme modelovat co nejpřesněji.

Toto modelování se využívá především v technických oborech, tj. např. při konstrukci dopravních prostředků.

Předpovídání chování

Ne vždy systému dobře rozumíme. Nemáme nad ním kontrolu a nemůžeme ho řídit. Při tomto modelování nás zajímá, jak se systém bude vyvíjet v budoucnu. Mezi typické příklady patří počasí nebo také vývoj cen na burzách s akciemi. V těchto případech se snažíme co nejdetailněji zachytit chování tohoto systému pomocí modelu. Pro návrh takovéhohle modelu využíváme například data z historie.

Tento postup modelování nám tedy slouží k „předpovídání budoucnosti.“



(a) Ráno (b) Poledne (c) Odpoledne

Obrázek 6: Předpověď počasí

Porozumění

Díky modelování můžeme porozumět věcem, kterým moc nerozumíme. To znamená, že cílem simulace bude porozumět systému a pochopit jednotlivé zákonitosti. Model vytváříme podle našich předpokladů, jak tento model vypadá. Pomocí simulace zjistíme, do jaké míry naše předpoklady odpovídají realitě. U tohoto modelování není cílem modelovat chování systému přesně. Cílem je spíše pochopení základních principů reálného systému.

Tato metoda modelování a simulace se využívá u komplexních systémů.

Učení, trénink, zábava

Model může sloužit jako pomůcka pro výuku, ale také jako prostředek pro trénink v reálném systému, typickým příkladem jsou simulátory v autoškolách. Kde si student autoškoly nejprve zkusí řídit auto pomocí trenažéru a až potom, co získá základní zkušenosti, jak řídit automobil, sedne za volat skutečného vozu. V neposlední řadě nesmíme zapomenout na využití modelu pro zábavu, kde se jedná především o počítačové hry a stavebnice.

3 Diskrétní simulace

Charakteristickou vlastností pro diskrétní simulaci je, že se proměnné modelu a čas mění skokově. Skokově znamená nespojitě. K těmto skokům dochází pouze, když nastane nějaká pro nás významná událost. Z této vlastnosti vyplývá, že skoky mají proměnnou délku. Skok, jinými slovy časový krok, má vždy velikost nejmenší hodnoty ze všech předem známých událostí, které se mají vykonat. Tímto časovým krokem dojde vždy k posunutí aktuálního času. V aktuálním čase se vždy provedou všechny události, které se mají v tomto čase vykonat. Pokud je to možné, dáno vlastností modelu, se vygenerují další události na nové časy a pokračuje se krokem, kdy se vybírá nejmenší hodnota skoku.

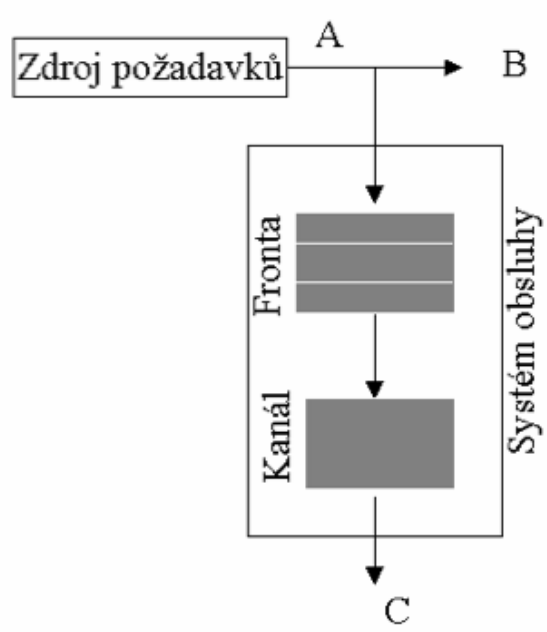
Pro lepší pochopení si uvedeme příklad jednoduchého systému hromadné obsluhy, ale nejdříve si pojem systém hromadné obsluhy vysvětlíme.

3.1 Systém hromadné obsluhy

Systém hromadné obsluhy, dále pak jen SHO, je systém obsahující obsluhové zařízení, které poskytuje obsluhu. Do tohoto systému vstupují entity, které požadují tuto obsluhu. Entitou můžou být například lidé, ale i neživé věci. Těmto entitám se většinou říká požadavky na obsluhu. Jednotlivé požadavky na obsluhu trvají stanovenou dobu. Po uplynutí této stanovené doby se obsluha uvolní a realizovaný požadavek odchází ve výstupní proud. Pokud je obsluhující zařízení obsazeno, řadí se požadavek na obsluhu do fronty.

SHO se typicky skládá z:

- vstupního proudu
- front
- kanálů obsluhy
- výstupního proudu



Obrázek 7: Systém hromadné obsluhy

Pramen:
http://agent-base-models-repast.googlecode.com/svn-history/r196/trunk/dp/SHO_1a.pdf

A - vstup požadavků do sytému

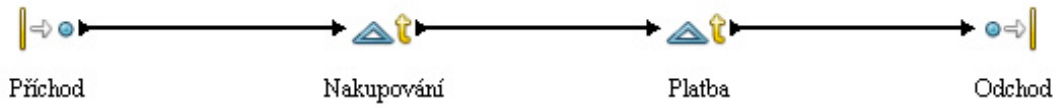
B - odmítnuté požadavky

C - realizované požadavky

Co sledujeme při simulaci:

- informace o časovém průběhu transakce procházející systémem
- doby čekání ve frontách
- zatížení obslužných linek

Mějme obchod, kde náhodně chodí zákazníci a podle potřeby nakupují. To znamená, že v obchodě stráví různou dobu, protože zákazníci pravděpodobně nebudou nakupovat to stejné zboží a tudíž stráví různou dobu nakupováním a je tedy zřejmě, že i k pokladně, kde platí, přicházejí náhodně. Pro jednoduchost budeme uvažovat, že v obchodě mají zatím pouze jednu pokladnu. Zákazníci taktéž stráví náhodnou dobu u pokladny, protože každý má jinak velký nákup. Doba obsluhy a intervaly mezi příchody jednotlivých zákazníků mají tedy charakter náhodné veličiny. Jelikož se jedná o simulaci diskrétní, nezajímá nás, jak se zákazníci v obchodě pohybují, ale zajímají nás pouze důležité události. V našem případě mezi důležité události patří: příchod zákazníka do obchodu, nakupování, platba u pokladny a odchod zákazníka.



Obrázek 8: Příklad diskrétní simulace

Pramen: Wikipedia
http://cs.wikipedia.org/wiki/Diskrétn%C3%AD_simulace

U těchto významných událostí můžeme zaznamenávat různé charakteristiky. Mezi které bychom mohli zařadit: jak dlouho zákazník nakupuje, jaká bude průměrná délka fronty, jakou dobu stráví zákazník v této frontě, kolik zákazníků pokladna (pokladní) obslouží atd. Zákaznici jsou v tomto případě entitou. Entity před spuštěním simulace mohou být v systému vloženy, nebo mohou během simulace přicházet. Entity dále vykonávají nějaké aktivity (např. nakupování , platby). Můžou systém opustit nebo v něm zůstat. Na začátku tohoto odstavce jsme si nastavili podmínku, že v obchodě máme pouze jednu pokladnu. Dále si tuto podmínku upřesníme. Budeme předpokládat, že tato pokladna může obsloužit pouze jednoho zákazníka. Jestliže zákazník přistoupí k pokladně mohou nastat tyto možnosti:

a) U pokladny není žádný zákazník, a tudíž může být zákazník ihned pokladnou obsloužen.

b) U pokladny už je zákazník. Nově příchozí zákazník se řadí do fronty před pokladnu.

V případě, že zákazník, který byl obsluhován je obsloužen, může nastat jedna z těchto situací:

a) U pokladny žádný zákazník nečeká, pokladna v tomto případě z pohledu simulace zůstává nečinná a čeká na příchod dalšího zákazníka.

b) U pokladny čeká jeden či více zákazníku. V tomto případě je zde vytvořena fronta čekajících zákazníků. Jeden ze zákazníků přistoupí k pokladně a začíná být pokladnou (pokladním) obsloužen. Jaký zákazník bude obsloužen je dáno vlastností fronty.

Režimy fronty:

a) FIFO – *First In First Out*, ten kdo přišel do fronty jako první, je obsloužen jako první.

b) LIFO – *Last In First Out*, ten kdo přišel do fronty jako poslední, je obsloužen jako první.

c) SIRO – *Select In Random Order*, z fronty se vybere náhodná entita.

d) podle priority (atributu)

3.2 Petriho sítě

Jednou z forem jak popsat diskrétní systém je použití petriho sítě.

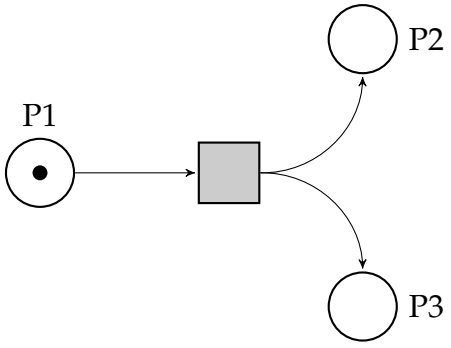
Definice Petriho sítě:

$$\sum = (P, T, F, W, C, M_0)$$

kde:

- P je množina míst
- T je množina přechodů P ⋂ T = Ø
- F ⊆ (P x T) ⋂ (T x P) incidenční relace
- W : F → {1, 2... } váhová funkce
- Kapacity míst C: P → N
- M_0 počáteční značení, M_0: P → N
- (M se nazývá značení Petriho sítě)

Nejčastější formou zadání Petriho sítě je pomocí grafu



- Místa - kružnice
- Přechody - čtverec
- Incidenční relace - šipky (orientované hrany)
- Váhové funkce - ohodnocení hran

Petriho sítě mohou modelovat: [2]

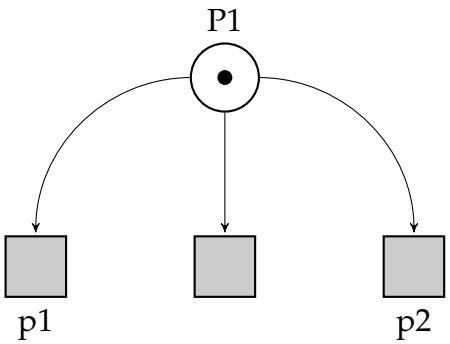
- paralelismus procesů
- komunikaci a synchronizaci
- nedeterminismus

Pro modelování diskrétních systémů zavádíme do klasických P/T Petriho sítí několik rozšíření — priority, pravděpodobnosti a doby přechodů.

Prioritní přechody

Z jednoho místa může vést více proveditelných přechodů. Těmto přechodům můžeme nastavit prioritu. Přechod s největší prioritou se provede.

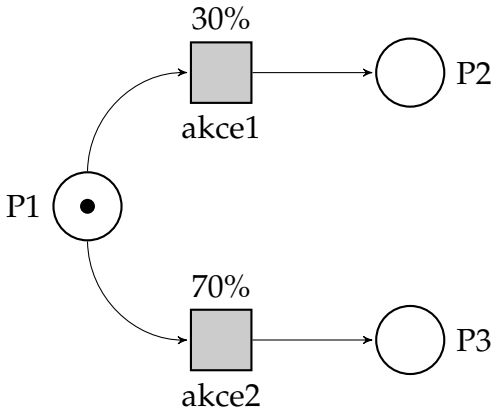
- $p_t \in \{0, 1, 2, 3, 4, 5, ...\}$ - čím větší číslo tím větší priorita
- defaulně je priorita rovna 0



p1=1, p2=2.

Pravděpodobnost provedení přechodu

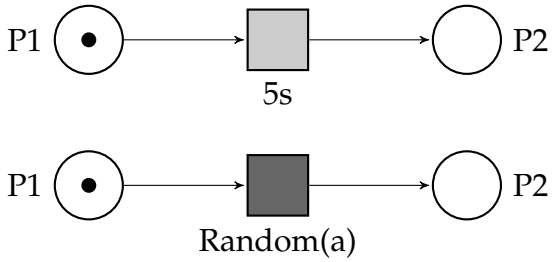
U přechodu stanovíme s jakou pravděpodobností se tento přechod provede.



S pravděpodobností 30% se provede akce1. Akce2 s pravděpodobností 70%.

Časované přechody

Tyto přechody jsou doplněny o modelový čas (parametr), který určuje dobu provádění přechodu. Čas může být konstantní nebo náhodně vygenerovaný.



4 DiscreteSimulationLibrary

4.1 Úvod

Simulační knihovna DiscreteSimulationLibrary je určena pro modelování a simulaci diskrétních systémů. Knihovna byla inspirována knihovnou SIMLIB a je kompletně naprogramovaná v jazyce C# s využitím .NET technologií. Což nám přináší řadu výhod. Mezi ně můžeme zařadit: objektově orientovaný jazyk, použití vláken a třídu monitor, která nám zjednodušuje synchronizaci a práci s vlákny. Knihovna DiscreteSimulationLibrary je tedy alternativou modelování a simulace diskrétních systému ke knihovně SIMLIB, která je napsána v jazyce C++, a je primárně určena pro operační systém LINUX.

4.2 Objektově orientovaná simulace

Objektově orientovaná simulace je založena na množině objektů, které mezi sebou komunikují, tj. posílají si zprávy. Každý systém můžeme rozdělit na jednotlivé objekty. Jak máme tento systém rozdělit je závislé na účelu modelu. V systému můžeme najít takové objekty, které mají shodné vlastnosti a podle toho je můžeme zařadit do jednotlivých tříd objektů. Každá třída definuje vnitřní strukturu těchto objektů, vlastní chování objektů v čase a reakce objektů na zprávy. Objekty provádějí akce, které jsou odezvou na zprávy. Akce dále mění stav objektu, které jsou dané jeho vnitřní strukturou. Jednotlivé akce jsou definované v objektu a souvisejí s popisem chování tohoto objektu. Tyto akce nazýváme metodami. Přijetí zprávy má za následek vyvolání jedné z metod, a podle toho jaká je vnitřní struktura této metody, může objekt na tuto zprávu reagovat.

Jak už jsme si popsali, různé objekty můžeme zařadit do různých tříd. Můžou nastat případy, kdy některé z tříd popisují vlastnosti objektů obecněji a některé jsou spíše konkrétnější. Tuto vlastnost můžeme využít k tomu, že třídy které mají obecnější charakter použijeme k popisu nějaké z konkrétnějších třídy. To tedy znamená, že zdědíme vlastnosti z obecné třídy, tyto vlastnosti můžeme modifikovat a dále je doplníme o vlastnosti, které nám tuto třídu zkonkretizují. Této hierarchii tříd říkáme dědičnost. Tato hierarchie tříd nám maximalizuje využití již existujících tříd, což má za následek zjednodušení a zpřehlednění modelu.

Příklad hierarchie tříd:

Dopravní prostředek
    Automobil
      Osobní automobil
      Nákladní automobil

Osoba
    Učitel
    Žák

4.2.1 Základní princip objektově orientovaného programování

Mezi základní principy objektově orientovaného programování (OOP) patří:

- Objekty
- Abstrakce
- Zapouzdření
- Skladování
- Delegování
- Dědičnost
- Polymorfismus

4.3 Struktura knihovny

Knihovna obsahuje základní třídy pro popis a chování diskrétních objektů, pro modelování standardních obslužných zařízení a pro sběr statistických údajů. Chování objektu lze popsat dvěma způsoby, buď pomocí událostí nebo procesů.

4.3.1 Simulární čas

V knihovně je simulární čas reprezentován proměnnou *Time*. Hodnota je typu double a její počáteční hodnota je 0. Uživatel hodnotu *Time* nemůže měnit, ale pouze zjistit.

4.3.2 Třída Process

Pro popis třídy objektů, které mají dynamické chování je v knihovně DiscreteSimulationLibrary připravena abstraktní třída *Process*. Tato třída implementuje rozhraní *IProcess*, která neobsahuje žádné metody, ale obsahuje pouze dvě proměnné:

- *IsWaiting*
- *Go*

Obě dvě tyto proměnné jsou typu bool.

Třída *Process* obsahuje metodu *Behavior*, která je taktéž abstraktní. Ta třída, která zdědí třídu *Process*, musí definovat chování objektu v čase. Toto chování se specifikuje právě v metodě *Behavior*. Metoda musíme přepsat a vložit do ní posloupnost příkazů, které se budou provádět.

```
public class Zakaznik : Process
{
    public Zakaznik(string name): base(name){}
    public override void Behavior()
    {
    }
}
```

Výpis 1: Dědění z třídy Process

Metoda *Behavior* nám tedy jasně popisuje, jak se bude objekt v čase chovat. Po spuštění objektu se začnou provádět jednotlivé příkazy, které se procházejí odshora dolů a postupně se vykonávají. Tato metoda může však obsahovat i příkazy, které způsobují čekání. To má za následek, že dojde k zastavení činnosti tohoto procesu v modelovém čase.

Mezi tyto příkazy patří metoda *Wait*. Ta přijímá jeden vstupní parametr, který je typu double. Tímto vstupním parametrem určíme, na jak dlouho proces pozastaví svou činnost. Jestliže bude v popisu chování u objektu O uveden příkaz *Wait(d)*, tak zůstane tento objekt O na dobu *d* neaktivní. To znamená, že jestli v modelovém čase *t* je proveden příkaz *Wait(d)*, dojde k obnovení činnosti tohoto objektu O v čase *t+d*. Je nutné podotknout, že každý proces je spuštěn na jiném vlákně. To nám přinesu tu výhodu, že jednotlivé procesy jsou na sobě nezávislé. Z této vlastnosti vyplývá, že když dojde k přerušení neboli pozastavení jednoho nebo více procesů, ostatní které nebyli pozastaveny dále vykonávají své příkazy.

Stavy pocesu

Jednotlivé procesy se v knihovně DiscreteSimulationLibrary vždy nacházejí v některém z těchto následujicích stavů:

- **Nový** – V tomto stavu se nachází proces, který byl vytvořen, ale nebyl ještě naplánován.
- **Pasivní** – Proces je pasivní, jestliže je jeho vlákno uspáno a čeká na probuzení. Jeho činnost je pozastavena.
- **Aktivní** – Proces je aktivní, pokud vykonává svou činnost.
- **Naplánovaný** – Proces právě neběží, ale má v plánovači naplánovanou událost. Poběží v jednom z nadcházejících simulačních kroků.
- **Ukončený** – Proces vykonal všechny své příkazy. Tento proces už nikdy nepoběží.

Uvedeme si příklad, jak proces prochází jednotlivými stavy. Prvním krokem je samotné vytvoření procesu. Předpokládejme, že jsme si vytvořili třídu *Zákazník*, která dědí z třídy *Process*. Jaká je základní konstrukce takovéto třídy jsme si ukázali v kapitole 4. Vytvoření provedeme klasickou inicializací objektu.

```
public class Program
{
    static  void Main(string[] args)
    {
        Zakaznik zakaznik = new Zakaznik("Zakaznik1") ;
    }
}
```

Výpis 2: Vytvoření procesu

Po vzniku se nachází proces ve stavu definovaném jako *Nový*. Odstartování procesu se provede metodou *Activate()*. Touto metodou, která je rovněž zděděná z třídy *Process*, se může sám objekt aktivovat. Po zavolání metody *Activate()* dojde k vytvoření nového vlákna a k naplánování spuštění procesu (odstartování vlákna) na aktuální čas *t*. V tomto okamžiku se proces nachází ve stavu *Naplánovaný*.

```
public class Program
{
    static  void Main(string[] args)
    {
        Zakaznik zakaznik = new Zakaznik("Zakaznik1") ;
        zakaznik.Activate()
    }
}
```

Výpis 3: Aktivace procesu

Aby byl proces ve stavu *Pasivní*, musí být jeho vlákno uspáno. To tedy znamená, že nevykonává žádnou činnost a čeká na probuzení. Jak už jsme si řekli, aby bylo vlákno uspáno neboli, aby proces nevykonával žádnou činnost, musí mít tento proces v popisu chování metodu *Wait* nebo některou z metod, které mají za následek uspání vlákna tohoto procesu. Proces je pasivní do doby, než nastane čas, ve kterém má být probuzeno. Další metody, které zapříčiní, aby byl proces ve stavu *Pasivní* jsou metody *Enter* a *Occupy*, které budou probrány v kapitolách 4.3.4, 4.3.5.

```
public class Zakaznik : Process
{
    public Zakaznik(string name): base(name)
    {
    }

    public override void Behavior()
    {
        Wait(time)
    }
}
```

Výpis 4: Ukázka popisu chování třídy Zákazník

Proces je ve stavu *Aktivní*, když vykonává svou činnost. Do chodu se tento proces dostane, až v momentě kdy je simulace spuštěna a proces má naplánovanou událost v plánovači. Jestliže nastane čas, kdy má být proces spuštěn, dojde k prouzení jeho vlákna.

Metoda *Behavior*, která popisuje chování procesu, musí vždy obsahovat na posledním místě metoda *Final*. Po tomto příkazu proces (objekt) zaniká a je ve stavu *Ukončený*.

Kvaziparalelní zpracování procesu

Jak už bylo popsáno výše, jednotlivé procesy jsou spuštěny na různých vláknech. Z této vlastnosti nám vyplývá, že v knihovně dochází k paralelnímu zpracování procesů. Nesmíme však opomíjet fakt, že běh simulace je spuštěn na jednoprocesorovém počítači. Z tohoto důvodu musíme přistoupit k takzvanému kvaziparalelnímu zpracování. U kvaziparalelního zpracování dochází v jednom časovém

okamžiku vždy ke zpracování jednoho procesu. Až tento proces ukončí svou činnost nebo je uspán, přichází na řadu další dosud uspaný proces.

4.3.3 Třída Event

Třída *Event* rovněž implementuje rozhraní *IProcess*. Ten objekt jehož nadtypem je třída *Event*, se zpravidla používá pro popis jednorázových dějů nebo dějů, které se periodicky opakují. Tato třída je abstraktní, a taktéž jako třída *Process* obsahuje přepisovatelnou metodu *Behavior*, v niž specifikujeme chování události. Rozdíl mezi třídou *Process* a *Event* je v tom, že třída *Event* je nepřerušitelnou a může sama sebe naplánovat a aktivovat na určitý čas. Podobně jako u procesu zde máme metodu *Activate*, která vytvoří nové vlákno pro tuto událost a naplánuje její spuštění na aktuální čas *t*. Typickým příkladem využití třídy *Event* je generátor, který periodicky vytváří instance procesů. Metoda *Activate* může přijímat i jeden vstupní parametr. Tímto vstupním parametrem je čas, kdy má být událost znovu aktivována. Metoda *Behavior* musí zase obsahovat jako poslední příkaz příkaz *Finish*.

```
public class Generator : Event
{
    public Generator(string name): base(name)
    {
    }

    public override void Behavior()
    {
    }
}
```

Výpis 5: Dědění z třídy Event

4.3.4 Třída Device

Třída Device je instantní třídou reprezentující obslužné zařízení. Jedná se o obslužné zařízení s *výlučným (exluzivním) přístupem*. Problém výlučného přístupu lze formulovat takto: Každý z *m* procesů systému požaduje takový přístup k abstraktnímu zařízení nebo zdroji *Z*, který vylučuje, aby v kterémkoliv okamžiku sdílel zařízení *Z* více, než jeden proces.[6]

```
Device pokladna = new Device("Pokladna 1");
```

Výpis 6: Příklad deklarace zařízení

Zařízení může být ve stavu kdy je obsazeno některým z procesů, nebo může být volné. Jaký je stav zařízení můžeme testovat metodou *IsOccupied*. Metoda má návratový typ bool. Jestliže je zařízení obsazeno vrátí metoda hodnotu TRUE, v opačném případě FALSE.

```
if (pokladna.IsOccupied())
{
    Console.WriteLine("Obsazeno");
}
```

Výpis 7: Příklad využití metody IsOccupied

Pro práci s zařízením jsou v třídě *Process* implementovány dvě metody (*Occupy, Release*). K obsazení zařízení voláme metodu *Occupy*. Metoda přijímá jeden vstupní parametr. Vstupním parametrem je reference na zařízení, které má být procesem obsazeno. Pro uvolnění zařízení slouží metoda *Release*. Přijímá pouze jeden vstupní parametr, a to referenci na proces, který zařízení obsadil.

```
Occupy(pokadna);
Wait(10);
Release(pokladna);
```

Výpis 8: Příklad použití metod Occupy a Release

V případě, že je zařízení některým z procesů využívané, zařadí se proces, který žádá o obsluhu do fronty. Tato fronta je součástí každého zařízení. Zařízení je schopné obsloužit pouze jeden požadavek na obsluhu. Uvolnit zařízení může pouze ten proces, který ho obsadil. Jestliže fronta obsahuje nějaké objekty (procesy) a došlo k uvolnění zařízení, obsadí se toto zařízení procesem, který je dán režimem fronty.

Režim fronty se určuje druhým parametrem v konstruktoru zařízení, při inicializaci objektu. Jaký režim fronty bude zvolen je dán hodnotami v rozsahu 0, 1, 2, 3. V případě, že tento parametr není zadán, je výchozím režimem fronty režim FIFO.

Režimy fronty u zařízení

- FIFO - 0
- LIFO - 1
- SIRO - 2
- dle priority procesu - 3

Co jednotlivé režimy znamená jsme si vysvětlili v kapitole 3.1

```
Device pokladna = new Device("Pokladna 1",1);
```

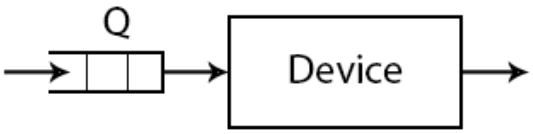
Výpis 9: Příklad deklarace zařízení s jiným režimem fronty

V momentě, kdy zařízení Z má nastaven režim fronty dle priority procesu, a v této frontě se nachází proces P1 s prioritou PR1 a proces P2 s prioritou PR2, můžou nastat tyto možnosti:

a) PR1$>$ PR2 $-$ proces P1 bude obsloužen jako první

b) PR1$<$ PR2 $-$ proces P2 bude obsloužen jako první

c) PR1$=$ PR2 $-$ bude obsloužen, ten který jako první přišel (FIFO)



Obrázek 9: Device

Jednotlivá zařízení, která jsou vytvořena v simulaci, uchovávají statistické údaje. Zařízení obsahuje metodu *Output*, pomocí které lze tyto statistické údaje vytisknout do konzoly. Statistickými informacemi jsou:

- kolik zařízení během simulace obsloužilo požadavků
- průměrná doba využití zařízení
- maximální délka fronty
- maximální doba čekání ve frontě
- minimální doba čekání ve frontě
- průměrná doba čekání ve frontě

4.3.5 Třída Store

Třída *Store* (sklad) je obdobou třídy *Device*. Charakteristickým prvkem je rovněž výlučný přístup. Rozdílem mezi třídou *Store* a *Device* spočívá v tom, že třída *Store* je schopna obsloužit více procesů najednou. U každého skladu určujeme, jaký maximální počet procesů může být současně obsloužen. V momentě kdy chce proces obsadit určitý počet jednotek a tento počet je menší než volné místo ve skladu, může proces obsadit požadovaný počet jednotek a volné místo se tímto zmenší. V případě, že by proces chtěl zabrat více jednotek ze skladu než momentálně disponuje, musí se zařadit do fronty a čekat do doby něž požadované místo bude volné. V tomto okamžiku je proces uspán. Princip výběru z fronty je totožný jako u zařízení (FIFO, LIFO, SIRO, priorita). Režim fronty se rovněž určuje v konstruktoru.

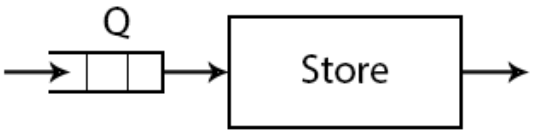
```
Store voziky = new Store("Voziky", 50);
```

Výpis 10: Příklad deklarace skladu

Pro základní operace se skladem jsou v třídě *Process* implementovány metody *Enter* a *Leave*. Metodou *Enter* je možno obsadit určitý počet jednotek. Ta přijímá dva parametry. A to jaký sklad bude obsazen a kolik jednotek bude zabráno. K uvolnění kapacity slouží druhá ze zmíněných metod a to metoda *Leave*. Ta má pouze jeden vstupní parametr. Parametr určuje, ze kterého skladu mají být uvolněny jednotky. Třída *Store* si pamatuje, kolik jednotek proces obsadil a podle tohoto počtu vrátí (uvolní) příslušný počet jednotek. Sklad dále obsahuje pomocné metody pro zjištění zda je prázdný (*Empty*), plný (*Full*) a kolik volných jednotek zbývá (*Free*).

```
Enter(voziky,2);
Wait(20);
Leave(voziky);
```

Výpis 11: Příklad použití metod Enter a Leave



Obrázek 10: Store

Tak jako zařízení uchovává sklad statistické informace. Výstup statistik lze provést metodou *Output*. Metoda *Output* tiskne do konzoly následující informace:

- jak velkou kapacitou sklad disponuje
- kolik procesů bylo obslouženo
- maximální použitá kapacita
- minimální použita kapacita
- průměrná použitá kapacita
- maximální délka fronty
- minimální délka fronty
- průměrná délka fronty
- průměrná doba čekání ve frontě
- maximální doba čekání ve frontě
- minimální doba čekání ve frontě

4.3.6 Třída RandomGenerator

Při simulaci diskrétních systémů je mnohdy zapotřebí modelovat náhodné jevy. Pro tento účel je v knihovně DiscreteSimulationLibrary implementována třída RandomGenerátor, která generuje náhodné čísla pro různá rozložení. Tato třída je statická a tudiž nevytváříme žádné objekty této třídy.

Metody třídy RandomGenerator

`NextInt(int max)`
Vygeneruje náhodné číslo v intervalu od 0 po max s rovnoměrným rozložením. Návratová hodnota je typu int.

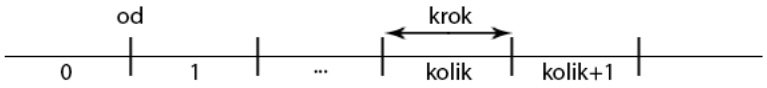
`NextInt(int min, int max)`
Vygeneruje náhodné číslo v intervalu od min po max s rovnoměrným rozložením. Návratová hodnota je typu int.

`NextDouble(double min = 0, double max = 1)`
Vygeneruje náhodné číslo v intervalu od min po max s rovnoměrným rozložením. V případě, že nezadáme vstupní parametry vrátí náhodné číslo v intervalu 0 až 1. Návratová hodnota je typu double.

`NextExp(double mean)`
Generuje exponenciální rozložení se střední hodnotou mean. Návratová hodnota je typu double.

4.3.7 Třída Histogram

Třída *Histogram* zaznamenává četnost zapisovaných hodnot v zadaných intervalech. U třídy zadáváme jméno histogramu, od jaké hodnoty začínáme, jak velký je skok neboli šířka intervalu. Šířka by měla být správně zvolena, neboť nesprávně zvolená šířka nám může snížit informační hodnotu. Poslední zadávánou hodnotou je kolik kroků se má udělat.



Deklarace může vypadat takto:

```
Histogram h = new Histogram("Histogram", 0, 10, 20);
```

Výpis 12: Ukázka deklarace histogramu

V tomto konkrétním případě se jedná o histogram jménem Histogram, který začíná od hodnoty 0, jeho interval má velikost 10 a celkový počet intervalů je 20.

Přidávání záznamů do histogramu se prování pomocí metody *AddRecord*. Má jeden vstupní parametr, který slouží pro zadání hodnoty.

```
h.AddRecord(10);
```

Výpis 13: Přidání záznamu do histogramu

Pro výstup histogramu slouží metoda *Output*. Ta tiskne tabulku četností a statistické údaje:

- minimální vstupní hodnotu
- maximální vstupní hodnotu
- počet vstupních hodnot
- průměrnou hodnotu hodnot
- směrodatnou odchylku hodnot

4.3.8 Třída Statistic

Pro záznam statistických údajů, je v knihovně DiscreteSimulationLibrary implementována třída *Statistic*.

Třída uchovává tyto údaje:

- maximální vstupní hodnotu
- minimální vstupní hodnotu
- počet zaznamenaných hodnot
- průměrnou hodnotu hodnot
- směrodatnou odchylku hodnot

Pro vložení hodnoty se používá metoda *AddValue*. Má jeden vstupní parametr a to velikost hodnoty. Metodou *Output* se tisknou statistické údaje.

```
Statistic stat = new Statistic();
for (int i = 0; i < 10; i++)
{
    stat.AddValue(i)
}
sta.Output();
```

Výpis 14: Příklad použití metody Statistic

4.3.9 Třída TStatistic

Objekty třídy TStatistic slouží pro záznam časových intervalů v systému. Pro zadání tohoto intervalu obsahuje třída dvě metody. Těmito metodami jsou, metoda *StartTime* a *EndTime*. Obě metody mají dva vstupní parametry. Prvním je, který z procesů zadává startovací resp. koncový čas a druhý parametr udává hodnotu těchto startovacích a koncových časů.

```
TStatistic tstat = new TStatistic();
tstat.StartTime(proces, 10);
...
tstat.EndTime(proces, 20);
```

Výpis 15: Příklad použití metody TStatistic

Třída *TStatistic* uchovává tyto informace:

- maximální velikost vstupního intervalu
- minimální velikost vstupního intervalu
- počet vstupních intervalů
- průměrnou hodnotu vstupních intervalů
- směrodatnou odchylku intervalů

Pro výstup těchto hodnot voláme metodu *Output*.

4.3.10 Třída Scheduler

Třída *Scheduler*, neboli plánovač je řídící třídou simulace. Tato třída zahrnuje veškeré metody pro naplánování a mazání procesů, probouzení jednotlivých vláken, rušení procesů a metodu pro spuštění simulace. Třída je z pohledu uživatele nezajímavá, protože je primárně určena pro samotný běh simulace. Jedinou podstatnou věcí pro uživatele je, že třída implementuje metodu (*Run*) a že využívá návrhový vzor Singleton. V každé simulaci, kterou bude uživatel pomocí této knihovny vytvářet, musí inicializovat objekt této třídy.

```
Scheduler es = Scheduler.Instance;
es.Run();
```

Výpis 16: Příklad použití třídy Scheduler

4.4 Příklad diskrétní simulace v knihovně DiscreteSimulationLibrary

Pro názornou ukázku, jak použít knihovnu DiscreteSimulationLibrary budeme pokračovat v příkladu, kdy zákazníci chodí náhodně do obchodu a nakupují. Systém nebude tak jednoduchý jako v kapitole 3.1, ale bude rozšířen o různé vlastnosti a funkcionality.

Definice systému bude vypadat takto:

Zákazníci přicházejí náhodně do obchodu s exponenciálním rozložením se střední hodnotou 5. Generování těchto zákazníků bude končit v momentě, kdy počet vygenerovaných zákazníků bude roven 100. Každý zákazník si bere nákupní košík a jde nakupovat. V obchodě je k dispozici 50 nákupních košíků. Zákazník nakupuje náhodnou dobu s normálním rozložením v intervalu od 10 do 15. V obchodě je oddělení lahůdek, které využije 30 % zákazníků, kteří navštíví tento obchod. V oddělení lahůdek zákazník nakupuje náhodnou dobu s exponenciálním rozložením a střední hodnotou 2. Oddělení lahůdek je schopno obsloužit dva zákazníky současně. V momentě kdy zákazník ukončí nákup, jde k jedné ze tří pokladen, které jsou v obchodě. Zákazník chce čekat nejkratší dobu a tudíž si vybírá pokladnu, u které je nejmenší fronta. V tomto modelu neuvažujeme závislost, že ten zákazník, který strávil v obchodu více času toho nakoupil více. Délka platby u pokladny bude v tomto případě taktéž náhodná s exponenciálním rozložením se střední hodnotou 3. Po zaplacení zákazník vrátí košík a odchází. Ze statistického pohledu budeme chtít zaznamenávat histogram, jakou dobu zákazník stráví v obchodě a všechny statistické informace jednotlivých zařízení a skladů.

V první řadě musíme do nového projektu importovat knihovnu DiscreteSimulationLibrary příkazem `using DiscreteSimulationLibrary`. Poté se pustíme do samotného vytváření všech objektů a tříd v systému. Postupně vytvoříme zařízení respektive sklady, které budeme potřebovat. Pro reprezentaci košíků použijeme třídu *Store* s kapacitou 50. Objekty všech tříd budeme vytvářet v metodě *main*.

```
static void Main(string[] args)
{
    Store voziky = new Store("Voziky", 50);
}
```

Dále budeme potřebovat, dle zadání, tří pokladny. Každá pokladna bude reprezentována třídou *Device*. Pro jednoduší implementaci vytvoříme seznam těchto pokladen.

```
List<Device> pokladny = new List<Device>();
for (int i = 1; i <= 3; i++)
{
    Device pokladna = new Device("Pokladna "+i);
    pokladny.Add(pokladna);
}
```

Oddělení lahůdek reprezentuje rovněž třída *Store*. Lahůdku současně obslouží dva zákazníky, tudíž třída *Store* bude mít kapacitu 2.

```
Store lahudky = new Lahudky("Lahudky", 2);
```

Samotný zákazník bude bázovou třídou *Process*. Zákazníkovi musíme předat reference na pokladny a lahůdky, které může využívat. Pro záznam statistik předáme referenci na objekt histogramu, který rovněž vytvoříme klasickou deklarací. Tyto reference předáváme v konstutoru. V neposlední řadě musíme popsat samotné chování zákazníka v metodě *Behavior*.

```
public class Zakaznik : Process
{
    List<Device> pokladny;
    Store voziky;
    Store lahudky;
    Histogram celk;
    public Zakaznik(string name, List<Device> pokladny, Store voziky, Store lahudky, Histogram
        celk): base(name)
    {
        this.pokladny = pokladny;
        this.voziky = voziky;
```

```
        this.lahudky = lahudky;
        this.celk = celk;
    }

    public override void Behavior()
    {
    double prichod = Time;                          \\zaznam prichodu zakaznika

    Enter(voziky, 1);                               \\obsazeni jednoho voziku

     if (RandomGenerator.NextDouble() <= 0.30)      \\30 % navstivi Lahudky
     {
        Enter(lahudky, 1);                          \\ obsazeni jedne pozice z Lahudek
        Wait(RandomGenerator.NextExp(2));           \\ exponencialni doba nakupovani
        Leave(lahudky);                             \\ uvolneni lahudek
    }

     Wait(RandomGenerator.NextDouble(10,15));       \\nakupovani v intervalu <10,15>

    int pokladna = 0;

    for (int i = 1; i < pokladny.Count; i++)                    \\vyber pokladny
        if (pokladny[pokladna].QueueLen > pokladny[i].QueueLen)
        pokladna = i;

    Occupy(pokladny[pokladna]);                     \\obsazeni pokladny
    Wait(RandomGenerator.NextExp(3));               \\cas platby
    Release(pokladny[pokladna]);                    \\uvolneni polkladny

    Leave(voziky);                                  \\vraceni voziku

    celk.AddRecord(Time - prichod);                 \\pridani zaznamu do histogramu
    Finish();                                       \\konec chovani zakaznika
    }
}
```

Pro generování zákazníků použijeme třídu *Event*. Jak už bylo napsáno v kapitole 4.3.3, třída je vhodná pro periodicky se opakující děje a může sama sebe aktivovat na určitý čas. V konstruktoru předáváme všechny potřebné reference na objekty,které potřebujeme.

```
class Generator : Event
    {
        List<Device> pokladny;
        Histogram celk;
        Store voziky;
        Store lahudky;
        int i = 1;

        public Generator(List<Device> pokladny, Store voziky, Store lahudky, Histogram celk)
        {
            this.pokladny = pokladny;
            this.voziky = voziky;
            this.lahudky = lahudky;
            this.celk = celk;
        }

        public override void Behavior()
        {
            if (i <= 100)                          \\podminka pro generovani 100 zakazniku
            {
                new Zakaznik("Zakaznik " + i, pokladny, voziky, lahudky, celk).Activate();
                                                   \\ aktivovani zakaznika na aktualni cas t
                Activate(Time + RandomGenerator.NextExp(2));
                                   \\aktivovani sama sebe na cas t + nahodne cislo
                i++;
            }
            else
            {
                Finish();       \\konec chovani generatoru
            }
        }
    }
```

Výpis 17: Generátor zákazníků

```
public class Program
{
    static void Main(string[] args)
    {
        Scheduler es = Scheduler.Instance;
        Store voziky = new Store("Voziky", 50);
        Store lahudky = new Store("Lahudky", 2);
        Histogram celk = new Histogram("Doba nakupovani v obchode", 0, 10, 10);
        List<Device> pokladny = new List<Device>();
        for (int i = 1; i <= 3; i++)
        {
            Device pokladna = new Device("Pokladna "+i);
            pokladny.Add(pokladna);
        }

        new Generator(pokladny, voziky, lahudky, celk, sc).Activate(); \\aktivovani generatoru
        es.Run();                                                      \\spusteni simulace

        \\tisk statistickych informaci
        foreach (Device pokladna in pokladny)
                pokladna.Output();
        voziky.Output();
        lahudky.Output();
        celk.Output();
    }
}
```

Výpis 18: Metoda main diskrátního modelu obchodu

4.4.1 Výstup statistických informací

HISTOGRAM Doba nakupovani v obchode

STATISTIC

Min = 10,48 Max = 30,87
Number of records = 100
Average value = 18,16
Standard deviation = 4,92

from	to	n	rel	sum
0	10	0	0,00000	0,00000
10	20	70	0,70000	0,70000
20	30	27	0,27000	0,97000
30	40	3	0,03000	1,00000
40	50	0	0,00000	1,00000
50	60	0	0,00000	1,00000
60	70	0	0,00000	1,00000
70	80	0	0,00000	1,00000
80	90	0	0,00000	1,00000
90	100	0	0,00000	1,00000

DEVICE Pokladna 1

Status - not BUSY
Number of request = 60
Average time utilization = 2,56

DEVICE - Queue

Incoming 33
Outcoming 33
Current length = 0
Maximal length = 2
Minimal time = 0,04
Maximal time = 8,38
Average time = 2,50

DEVICE Pokladna 2

Status - not BUSY
Number of request = 26
Average time utilization = 3,83

DEVICE - Queue

Incoming 13
Outcoming 13
Current length = 0
Maximal length = 2
Minimal time = 0,22
Maximal time = 15,16
Average time = 4,58

DEVICE Pokladna 3

Status - not BUSY
Number of request = 14
Average time utilization = 3,79

DEVICE - Queue

Incoming 6
Outcoming 6
Current length = 0
Maximal length = 1
Minimal time = 0,00
Maximal time = 8,54
Average time = 4,42

STORE Lahudky

Capacity = 2
Number of enter operation = 30
Minimal use capacity = 1
Maximal use capacity = 2
Average use capacity = 1,53

STORE - Queue

Incoming = 2
Outcoming = 2
Current length = 0
Maximal length = 2
Minimal time = 0,40
Maximal time = 2,08
Average time = 1,24

STORE Voziky

Capacity = 50
Number of enter operation = 100
Minimal use capacity = 1
Maximal use capacity = 16
Average use capacity = 9,59

STORE - Queue

Incoming = 0
Outcoming = 0
Current length = 0
Maximal length = 0
Minimal time = 0,00
Maximal time = 0,00
Average time = 0,00

5 Spojitá simulace

Spojitá simulace je charakteristická tím, že všechny stavové proměnné nabývají hodnoty z nějakého definovaného intervalu a v průběhu času se mění spojitě. K popisu chování spojitého modelu se využívají soustavy diferenciálních rovnic, ať už obyčejných nebo parciálních. Jelikož k popisu tohoto chování používáme soustavy diferenciálními rovnicemi, musíme tyto soustavy řešit numericky. U spojité simulace je simulární čas diskretizován, to znamená, že čas je rozdělen na jednotlivé intervaly, kde velikost tohoto intervalu je dána integračním krokem. Hodnota simulárního času se tedy zvyšuje o tento integrační krok. Integrační krok musí být zvolen tak, aby splňoval požadavek na přesnost výpočtu.

Aplikace spojitých simulačních modelů přinášení následující problémy: [7]

- výběr vhodné metody numerické integrace,
- délka integračního kroku s ohledem na požadovanou přesnost,
- vysoké nároky na spotřebu strojového času,

Typické příklady řešené pomocí spojitých simulačních modelů: [7]

- kmitání mechanických systémů,
- řešení elektrických a elektronických systémů,
- kompartmentové systémy,
- dynamika populací

5.1 Metody numerické integrace

Jak už jsem si řekli, abstraktní model spojité simulace je obecně popsán soustavou diferenciálních rovnic. Z velkého množství diferenciálních rovnic dovedeme explicitně řešit jen velmi malou část z nich. Z tohoto důvodu jsme nuceni řešit tyto diferenciální rovnice pomocí přibližných metod. V této kapitole se budeme těmito metodami zabývat. Pomocí přibližných metod hledáme numerické řešení jen na zvolené množině bodů v daném intervalu. Interpolací z těchto hodnot pak můžeme najít přibližné hodnoty řešení také pro ostatní body ze zvoleného intervalu. Jedná se o Eulerovu polygonální metodu a metodu Runge-Kuttovu. [8]

5.1.1 Základní pojmy

Uvažujme pro jednoduchost soustavu n obyčejných diferenciálních rovnic 1.řádu

$$\frac{dy_i}{dt} = f_i(t, y_1, y_2, ..., y_n) \tag{1}$$

s počáteční podmínkou $y_i = y_{i0}$, kde $i = 1, 2, ..., n$, tj. budeme řešit *Cauchyho úlohu*.

Tuto úlohu budeme zapisovat v maticovém tvaru. Nechť G je podmnožina euklidovského prostoru R^2, buď f reálná funkce definovaná na G.

$$y' \equiv \frac{dy}{dt} = f(t, y) \tag{2}$$

s počáteční podmínkou $y(t_0) = y_0$, kde $[t_0, y_0] \in G$.
[8] [7]

5.1.2 Princip numerických metod

Při numerickém řešení Cauchyho úlohy na intervalu $< a, b >$ postupujeme tak, že zvolíme konečnou množinu bodů $t_i, i = 1, 2, ..., n$, takových, že

$$a = t_0 < t_1 < t_2 < ... < t_n - 1 < t_n = b.$$

Tuto množinu nazýváme *síť* a jejich prvky *uzly*. [8]

Numerickým řešením počáteční úlohy v tomto případě bude posloupnost $\{y_i\}$ hodnot

$$y_0 = y(t_0), y_1 = y(t_1), ..., y_n = y(t_n)$$

Hodnoty v intervalu $< t_0, t_n >$ jsou nezávislé a z pravidla nám určují čas. Hodnota výrazu $h = t_{i+1} - t_i$ přitom udává velikost integračního kroku (*krok sítě*). [7]

Při konstruování numerických metod vycházíme z přírůstku zobrazení

$$y_{i+1} = y_i + \Delta y_i = y_i + h_i S(t_i, y_i, h_i), \tag{3}$$

kde $S(t_i, y_i, h_i) = \frac{\Delta y_i}{h_i}$ je směrnice přímky určená body $(t_i, y_i), (t_{i+1}, y_{i+1})$, proto funkci S nazýváme přírůstkové zobrazení nebo směrová funkce. [8]

Metodu danou vztahem (3) nazýváme jednokroková metoda, protože počítaná hodnota y_{i+1} závisí jen na y_i. [8]

Při numerickém řešení se dopouštíme kromě zaokrouhlovacích chyb i chyb způsobených diskretizací úlohy. Tyto chyby posuzujeme lokálně i globálně.[8]

Globální (akumulovanou) diskretizační chybou v uzlu t_i nazýváme rozdíl

$$e_i = y(t_i) - y_i.$$

Globální chyba je výsledkem diskretizačních chyb z předcházejících kroků. Je to tedy celková chyba po i-tém kroku způsobená metodou. Chybu způsobenou diskretizací v jednom kroku nazýváme lokální diskretizační chyba. Je to nepřesnost, s jakou splňují hodnoty přesného řešení y(t) v uzlech sítě rekurentní vztah (3). [8]

Nechť z(t) je přesné řešení úlohy

$$\frac{dy}{dt} = f(t, y),$$
$$y(t_{i-1}) = y_{i-1}$$

pak pro lokální chybu platí

$$d_i = z(t_i) - y_i.$$

Cílem numerických metod je nalezení numerického řešení. Od každé numerické metody požadujeme, aby při zmenšování kroku, tj. $max(h_i) \to 0$, posloupnost numerických řešení konvergovala k přesnému řešení.

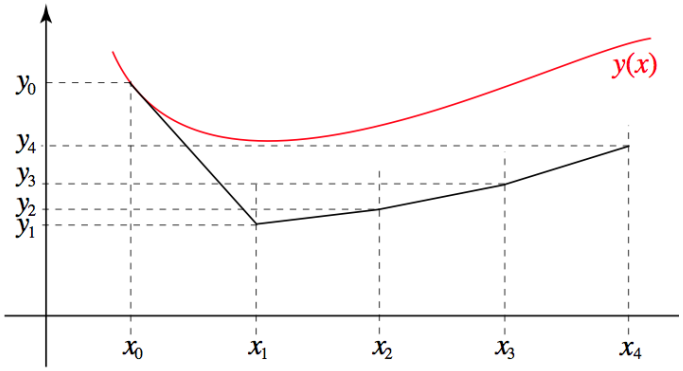
Kvalita použité numerické metody se hodnotí podle těchto základních kritérií: [7]

- přesnost metody (velikost lokální chyby a prostředky pro její odhad),
- stabilita metody,
- časová náročnost výpočtu,
- nároky na operační paměť počítače.

5.1.3 Eulerova metoda

Eulerova metoda je představitelem nejjednodušší jednokrokové metody, ale je však obvykle nejméně přesná, kde nový stav lze stanovit z předcházejícího na základě vztahu

$$y_{i+1} = y_i + hf(x_i, y_i). \tag{4}$$



Pramen: HASÍK, K.: Numerické metody

Uvedený vztah lze jednoduše získat dvěma způsoby. [8][10]

1. Nahrazení vztahu (2) diferenčním vztahem

   $$y'(t_n) = \frac{y_{n+1} - y_n}{h} = f(t_n, y_n)$$

   odkud ihned plyne Eulerův vztah.

2. Vztah pro Eulerovu metodu můžeme také získat aproximací hodnoty y_{i+1} Taylorovým polynomem funkce y v bodě x_i. Pro dvakrát spojitě diferencovatelnou funkci y dostáváme

   $$y_{i+1} = y_i + hfy' + \frac{1}{2}h^2y''(\xi)$$

   kde $\xi \in < t_i, t_i + 1 >$. Poslední člen zanedbáme, zároveň získáme lokální chybu Eulerovy metody

   $$d_i = \frac{1}{2}h^2y''(\xi) = O(h^2)$$

Příklad 1. Řešte pomocí Eulerovy metody soustavu diferenciálních rovnic.

$$x' = xy,$$

$$y' = x - y,$$

které vyhovují počáteční podmínce $x(0) = 1, y(0) = 2$, na intervalu$< 0, 2 >$ s počtem kroků 4.

Řešení. Přibližnou hodnotu řešení hledáme pomocí iteračního vzorce

$$x_{i+1} = x_i + h \cdot f(x_i, y_i),$$

$$y_{i+1} = y_i + h \cdot g(x_i, y_i),$$

kde určíme krok $h = \frac{2-0}{4} = 0.5$, přičemž $x_0 = 1, f(x_i, y_i) = x_i y_i, g(x_i, y_i) = x_i - y_i$.

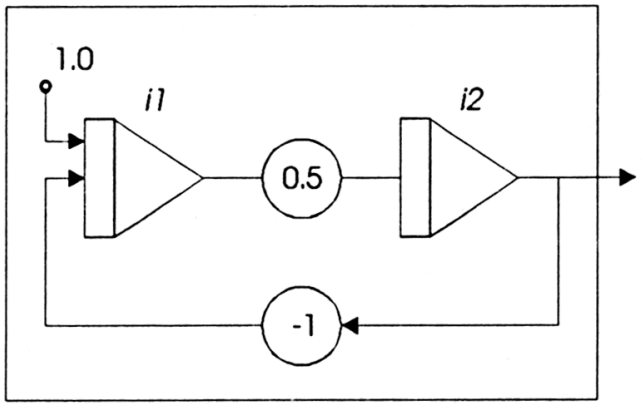
Jednotlivé kroky výpočtu budeme zaznamenávat do tabulky. V prvním sloupci je pořadí kroku i, ve druhém sloupci je bod t_i, ve kterém hledáme aproximaci řešení, ve třetím sloupci je x_i, aproximace hodnoty $x(t_i)$, ve čtvrtém sloupci je y_i, aproximace hodnoty $y(t_i)$, v pátém sloupci je „pomocný" výpočet $h \cdot f(x_i, y_i)$ a v šestém sloupci je „pomocný" výpočet $h \cdot g(x_i, y_i)$.

$\mathbf{i}$	$\mathbf{t_i}$	$\mathbf{x_i}$	$\mathbf{y_i}$	$\mathbf{0.5 \cdot (x_i \cdot y_i)}$	$\mathbf{0.5 \cdot (x_i - ty_i)}$
0	0	1	2	1	-0.5
1	0.5	2	1.5	1.5	0.25
2	1	3.5	1.75	3.06	0.88
3	1.5	6.56	2.62	8.61	1.97
4	2	15.18	4.59		

Tabulka 1: Tabulka kroků s jednotlivými výpočty

6 ContinuousSimulationLibrary

Pro popis chování spojitého modelu byla vytvořena knihovna *ConinuousSimulationLibrary*. Toto chování se v knihovně popisuje propojením objektů, které reprezentují integrátory. Propojení objektů se realizuje zadáním matematického výrazu při vytváření objektu třídy *Integrator*. Pro zadávání výrazů byla využita knihovna NCal, která disponuje třídou *Expression*, pomocí niž výraz zadáváme.



Obrázek 11: Příklad spojitého bloku

Pramen: Skripta VUT Brno: Modelování a simulace

6.1 Struktura knihovny

6.1.1 Třída Expression

Pro zadávání výrazu (diferenciální rovnice) do třídy *Integrator* jsme použil již existují knihovnu *NCal*. Knihovna obsahuje třídu *Expression*, pomocí níž může jednoduše zadavat diferenciální rovnici. Tuto rovnici zadáváme jako řetězec, tedy v jazyce C# jako string.

Příklad 2. Chceme zadat pomocí třídy *Expression* rovnici:

$$x = y - g,$$

kde x, y jsou proměnné a g je konstanta s hodnotou 9.81.

```
Expression x = new Expression("y−g");
x.Parameters["g"] = 9.81;
```

Výpis 19: Příklad použití třídy Expression

6.1.2 Třída Integrator

Třída *Integrator*, jak už její název napovídá, reprezentuje samotný integrátor (v obrázku 11, jako i1, i2). U této třídy využíváme dvě základní vlastnosti.

- nastavení počáteční podmínky,
- proměnnou *Value*, která vrací hodnotu objektu.

Inicializace objektu *Integrátor* se provádí standardním způsobem, kde v konstruktoru nastavujeme tři parametry.

1. jméno integrátoru − jméno musí být shodné s identifikátorem tohoto objektu
2. výraz (diferenciální rovnici)
3. počáteční podmínku

```
Integrator x = new Integrator("x", new Expression("x∗y"), "x=1;y=2") ;
```

Výpis 20: Příklad inicializace Integratoru

Při inicializaci se z počáteční podmínky nastaví hodnota *Value*. Tato hodnota se během simulace mění podle výpočtu z diferenciální rovnice, která byla zadána. Třída dále obsahuje metodu *AllValuesInTime*, která vrátí slovník, který obsahuje všechny časy a k nim odpovídající hodnoty.

6.1.3 Třída Simulation

Třída *Simulation* je řídící třídou spojité simulace. Podobně jako *Scheduler* u diskrétní simulace, zahrnuje veškeré metody pro nastavení parametrů a běh simulace. Interval běhu simulace se nastavuje metodou *Init*. Metoda *Init* přijímá dva vstupní parametry typu double. Prvním je hodnota začátku simulace a druhým je hodnota konce simulace. Pro nastavení velikosti kroku slouží metoda *SetStep*, která přijímá jeden vstupní parametr rovněž typu double. Parametrem určíme

velikost tohoto kroku. Jelikož je třída třídou řídící, obsahuje i soukromé metody, které jsou pro uživatele skryty. Mezi tyto metody patří metody pro výpočet numerické integrace. Numerická integrace je v knihovně reprezentována Eulerovou metodou. Jak pracovat s touto třídou si ukážeme v konkrétním příkladu 6.2.

6.1.4 Třída Graph

Třída *Graph* slouží pro vykreslení grafu ze záznamu hodnot spojité simulace. Tato třída obsahuje pouze jednu metodu a to metodu *Paint*. Metoda přijímá pět vstupních parametrů.

1. jméno grafu
2. název osy X
3. název osy Y
4. slovník, kde klíč určuje hodnotu osy X a hodnota klíče určuje hodnotu osy Y
5. velikost kroku zaznamenávaných hodnot

6.2 Příklad spojité simulace v knihovně ContinuousSimulationLibrary

Jako příklad spojité simulace budeme uvažovat systém kola automobilu. Experiment bude sledovat odezvu kola na jednotkový skok. Velikost skoku bude 0.001 a chování experimentu budeme sledovat v intervalu od 0 po 3. Výstupem simulace bude graf se jméne Kolo.png, osa X bude představovat čas, osa Y bude představovat hodnotu v. Rovnice, popisující tlumené kmitání kola:

$$Mx'' + Dx' + kx = F(t)$$

kde
v je rychlost pohybu kola,
y je výchylka kola z klidové polohy,
$F(t)$ je vstupní budicí funkce (např. jednotkový skok) a
k, D, M jsou konstantní parametry systému kola

Rovnici převedeme na soustavu diferenciálních rovnic prvního řádu

$$v' = \frac{F - Dv - ky}{M}$$

$$y' = v$$

Prvním krokem, který musíme udělat, je import knihovny do našeho projektu příkazem `using ContinuousSimulationLibrrary`. Poté bude následovat vytvoření všech objektů (*Kolo, Simulation, Integrator,...*) a nastavení veškerých parametrů. Všechny potřebné objekty budeme inicializovat v metodě *main*.

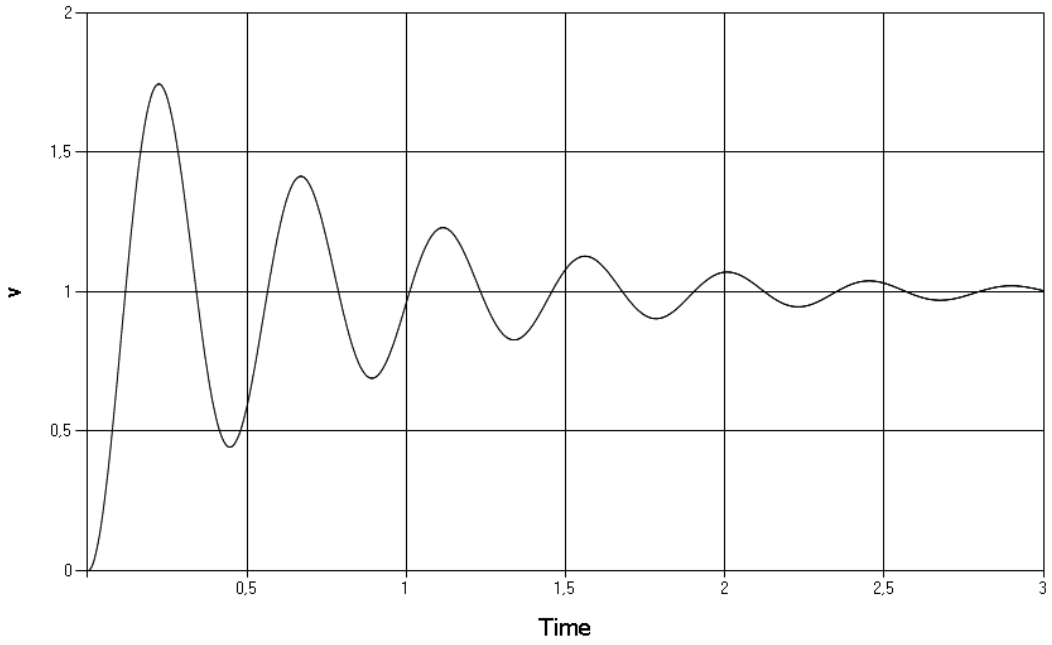
```
public class Kolo
    {
        Simulation s;
        Integrator v, y;
        public Kolo(Simulation s,double F,double M,double D,double k)
        {
            string initialCondition = "y=0;v=0";                    //pocatecni podminka
            Expression e = new Expression("((k*F)-(D*v)-(k*y))/M");
            e.Parameters["F"] = F;
            e.Parameters["M"] = M;
            e.Parameters["D"] = D;
            e.Parameters["k"] = k;
            this.s = s;
            this.v = new Integrator("v", e, initialCondition );
            this.y = new Integrator("y", new Expression("v"), initialCondition );
            s.AddIntegrator(v);                                     //pridani do ridici tridy integrator
            s.AddIntegrator(y);                                     //pridani do ridici tridy integrator
        }
        public void PaintGraph() //vykresleni grafu
        {
            Graph g = new Graph();
            g.Paint("Kolo.png", "Time", "Y", y.AllValuesInTime, 0.001);
        }
    }
```

Výpis 21: Vytvoření třídy Kolo

```
static void Main(string[] args)
        {

            Simulation s = Simulation.Instance;
            Kolo k = new Kolo(s,1,2,5.656,400);
            s.SetStep(0.001);
            s. Init (0, 3);
            s.Run();
            k.PaintGraph();
        }
```

Výpis 22: Metoda main spojitího modelu Kola



Obrázek 12: Graf tlumení kmitu kola

7 Závěr

Cílem této práce bylo vytvořit simulační knihovnu pro modelování a simulaci. Výsledkem mé práce jsou vytvořeny knihovny dvě, přičemž jedna je určena pro modelování a simulaci diskrétních systémů (DiscreteSimulationLibrary) a druhá je určena pro modelování a simulaci spojitých systémů (ContinuousSimulationLibrary). Obě knihovny jsou inspirovány knihovnou SIMLIB a jsou napsány v programovacím jazyce C# s využitím .Net technologií.

Samotný text práce popisuje jak teoretickou část, kde se čtenář seznámil se základními vlastnostmi a principy modelování a simulace diskrétních resp. spojitých systémů, tak část praktickou, která je primárně určena jako návod, jak s knihovnami pracovat a kde je využívat.

Knihovna ContinuousSimulationLibrary využívá nejjednodušší numerickou integraci a je určena pro jednoduché spojité systémy, kde nebude kladen velký důraz na přesnost.

8 Reference

[1] Kůs Zdeněk, *Simulace* [online]. Publikováno 10.10.2001 [cit. 2013-4-24]
Dostupné z:
www.kod.tul.cz/info_predmety/Psi/01-Simulace-S%20.pdf

[2] Peringer Petr, *Modelace a simulace* [online]. Publikováno 6.12.2012 [cit. 2013-4-24]
Dostupné z:
http://www.fit.vutbr.cz/study/courses/IMS/public/prednasky/IMS.pdf

[3] Dorda Michal, *Úvod do modelování a simulace systémů* [online]. [cit. 2013-4-24]
Dostupné z:
homel.vsb.cz/∼ dor028/Aplikace_2.pdf

[4] Palánek Radek, *Modelování a simulace komplexních systémů*, Nakladatelství Masarykovy univerzity, 2011.

[5] Cingel Viktor, *Modelovanie a simulácia na PC*, GRADA a.s, 1992.

[6] Rábová Z., Zendulka j., Češka M., Peringer P.,Janoušek V.,
Modelování a simulace, Nakladatelství VUT Brno, 1992.

[7] Ivan Křivý, Evžen Kindler, *Simulace a modelování*[online].[cit. 2013-4-24]
Dostupné z:
http://prf.osu.cz/kip/dokumenty/Msm.pdf

[8] Hasík Karel, *Numerické metody* [online]. [cit. 2013-4-24]
Dostupné z :
http://www.slu.cz/math/cz/knihovna/ucebni−texty/Numericke-metody/Numericke-metody.pdf

[9] Glynn J., Watson K., Skinner M., Robinson S., Nagel Ch., Allen K.S., Cornes O., Greenvoss Z., Harvey B., *C# Programujeme profesionálně*, COMPUTER PRESS.

[10] Jiří Macur, *Numerické metody* [online]. [cit. 2013-4-24]

Dostupné z:

http://www.fce.vutbr.cz/studium/materialy/Dynsys/kap7/kap7.htm#Eulerova%20metoda

[11] Simlib [online]. [cit. 2013-4-24]

Dostupné z:

http://www.fit.vutbr.cz/%7Eperinger/SIMLIB/.cs